Question Answering System Report

Introduction

In this project, I implemented a **Retrieval-Augmented Generation (RAG) system** for Question Answering (QA). The goal was to construct a system that, given a question, retrieves relevant passages from a corpus and extracts the most accurate answer. The system supports multiple retrieval methods and uses a fine-tuned reader model for answer extraction. This report details the experiment setup, methods used, evaluation approach, results, and potential improvements.

Experiment Environment

- **Python Version**: 3.11.2
- **OS**: Debian 6.1.128-1 (64-bit)
- **CPU**: Intel(R) Xeon(R) @ 2.20GHz (6 cores, 12 threads)
- **GPU**: NVIDIA A100-SXM4-40GB (CUDA 12.2, Driver 535.216.01)
- **RAM**: 83 GiB (Used: 5.3 GiB, Available: 78 GiB)

Reader: Model Choice & Fine-Tuning

I used **T5 (Text-to-Text Transfer Transformer)** as the reader model. T5 is an **encoder-decoder model** that can handle sequence-to-sequence tasks, making it highly flexible for QA. Unlike extractive models such as BERT, T5 generates answers in free text form, making it more adaptable to nuanced responses.

Why T5?

Pre-trained T5 models have been shown to perform well when finetuned on QA benchmarks, including SQuAD and Natural Questions. Unlike BERT-based models that highlight an answer span, T5 generates full responses.

However, as a generative model, T5 is more prone to errors if not properly fine-tuned. Optimizing hyperparameters could improve performance. While I used T5-small, larger variants (T5-base, T5-large) may produce better results. Additionally, comparing T5 with BERT/RoBERTa-based models could help assess whether generative QA models provide more accurate real-world answers.

Retrieval Methods Used

To retrieve relevant passages before answering, I implemented four retrieval methods. The code is modular, allowing for easy addition of new retrievers, making experimentation and extensibility easy.

1. Sparse Retrieval (BM25)

BM25 (Best Matching 25) is a sparse retrieval method that ranks documents based on term frequency (TF) and inverse document frequency (IDF). It assigns a score to each document by considering the presence and weight of query terms in the document. In this implementation, we use the bm25s library to tokenize and index the passages. The similarity score is computed using BM25 ranking, which measures lexical overlap between the query and documents. While BM25 is efficient for exact keyword matching, it lacks semantic understanding, making it less effective for queries requiring conceptual similarity matching.

2. Dense Retrieval (FAISS/ChromaDB)

Dense retrieval uses dense vector representations to find semantically similar passages to a given query. This implementation employs FAISS (Facebook AI Similarity Search) to store and search for embeddings efficiently. The passages are encoded using the all-MiniLM-L6-v2 transformer model from sentence-transformers. Retrieval is performed by computing the query's embedding and performing an approximate nearest neighbors (ANN) search using FAISS's IndexFlatIP, which ranks results based on inner product similarity (equivalent to cosine similarity for normalized vectors). This method is superior

to sparse retrieval for capturing semantic meaning but may struggle with rare or out-of-vocabulary words

3. Hybrid Retrieval (BM25 + Dense Search)

Hybrid retrieval combines sparse (BM25) and dense (FAISS) retrieval to maximize recall and improve document ranking. This implementation first retrieves results separately using BM25 (lexical matching) and FAISS (semantic similarity via all-MiniLM-L6-v2). The results are then merged using a weighted score fusion approach, where BM25 scores are weighted by a factor alpha and dense scores by (1 - alpha). Scores are normalized to prevent bias toward either method.

4. Contextual Retriever (ChromaDB, Self-RAG, etc.)

ChromaDB is a dense vector retrieval system optimized for large-scale search with persistent storage. This implementation uses the BAAI/bge-base-en embedding model to encode documents into dense vectors and stores them in ChromaDB. BGE is trained for retrieval, while MiniLM is for general sentence similarity, so BGE is maybe better for query-to-document matching. Queries are also encoded using the same model, and retrieval is performed via approximate nearest neighbors (ANN) search based on cosine similarity. Unlike the previous three,, ChromaDB offers persistent storage. ChromaDB has no built-in support for keyword-based retrieval (BM25), making it less effective for exact keyword matches.

Evaluation Approach

To avoid data leakage and simulate real-world conditions:

- Train Set (train-v2.0.json): Used for fine-tuning the reader (divided into train & eval splits).
- Dev Set (dev-v2.0.json): Used for final evaluation of both the fine-tuned t5 (as a test set) and the full RAG system.

Why This Evaluation Approach?

If your model only sees the train set, then when tested on the dev set, it must generalize to unseen passages. This ensures that the retrieval step is not artificially benefiting from a reader that has already memorized the answers. It also ensures a fair and meaningful comparison: Evaluating the reader alone gives an upper bound on performance. So the flow is as follows:

The model already has access to the exact passages (gold context). Run it directly on the SQuAD dev set. Evaluate the full RAG system (Retriever + Reader) This tests how well the retriever finds relevant passages. The reader now depends on retrieved context, not the gold passage. If retrieval is good, performance should be close to the reader-only setup.

Evaluation Metrics

- Exact Match (EM): Measures the percentage of answers that match exactly.
- F1 Score: Measures token overlap between predicted and ground-truth answers.

Expected Outcomes:

✅ If RAG performance ≈ Reader-only performance, retrieval is working well.
⚠️ If RAG performance << Reader-only performance, retrieval is weak—consider better retrieval models, reranking, or more passages per query.

Evaluation results (for the reader and all four RAG systems)

Method	EM	F1 Score	HasAns EM	HasAns F1	NoAns EM	NoAns F1
Reader	**53.21%**	**56.28%**	**55.01%**	**61.15%**	**51.42%**	**51.42%**
Sparse	52.75%	54.98%	41.67%	46.14%	63.80%	63.80%
Dense	51.94%	53.73%	31.17%	34.75%	72.65%	72.65%

Hybrid	52.75%	54.98%	41.67%	46.14%	63.80%	63.80%
ChromaDB	52.74%	54.77%	36.71%	40.78%	68.73%	68.73%

Conclusions based on the results

> Why does RAG seem to improve NoAns cases? Expected but somewhat artificial. In SQuAD, NoAns cases have passages that seem relevant but lack the required answer. The retriever, however, selects the most relevant document it can find, often different from the original NoAns passage. This gives the model new context, which may make it less confident in generating an incorrect answer, improving NoAns performance.
>
> Reader (Gold Context) performs best, as expected: it has access to the correct passage. Sparse retrieval works well for answerable cases, while dense retrieval improves NoAns cases (72.65% EM) but struggles with HasAns because semantic similarity can return related but less precise passages. Hybrid mirrors BM25, suggesting either that BM25 dominates the score fusion, or a bug. ChromaDB underperforms on HasAns compared to Sparse retrieval, likely due to embedding-based retrieval missing exact keyword matches.

Deployment

The system runs as a FastAPI service, handling QA requests over REST. The retrieval method (BM25, FAISS, Hybrid, ChromaDB) is set dynamically using RETRIEVER_TYPE in the make command.

A retriever cache (.cache/retriever_cache.pkl) speeds up startup, and the T5 model can load from either Hugging Face Hub or local checkpoint storage. Retrieval is done on the

SQuAD train set to see how the RAG is doing on a large collection of passages. ChromaDB works with both local storage and server-based retrieval.

Improvements & Future Work

- Add latency/throughput as metrics
- Reranking (cross encoder)
- Contextual Retrieval
- Model Quantization
- Alternative Retrieval Methods:
    - Self-RAG: Iterative retrieval refinement using feedback loops.
    - ElasticSearch: Combines BM25 and dense retrieval for scalability.
- Fine-Tuning & Error Analysis:
    - Tune hyperparameters for better performance.
    - Analyze model performance on answerable vs. unanswerable questions and WH-question types to see why the model is not performing well.
- Add asyncio for concurrent request handling and lower latency.
- Write tests.